

Board Games

By Mike Hahn

Copyright © 2008 Vecset.net

Date Last Edited: 15-Sep-2008

Table of Contents

Overview	1
<u>Introduction</u>	<u>1</u>
<u>Board Games Overview.....</u>	<u>1</u>
 Board Game Codes	2
<u>No-Code: Codeless Programming</u>	<u>2</u>
<u>Code 1: Automation</u>	<u>3</u>
<u>Code 2: Move Constraints.....</u>	<u>3</u>
<u>Code 3: Auto Move.....</u>	<u>4</u>
 Scramble	5
<u>Scramble Codeless</u>	<u>5</u>
<u>Scramble Automation</u>	<u>8</u>
<u>Scramble Move Constraints.....</u>	<u>8</u>
<u>Scramble Auto Move.....</u>	<u>9</u>
 Backgammon.....	9
<u>Backgammon Codeless</u>	<u>9</u>
<u>Backgammon Automation.....</u>	<u>10</u>
<u>Backgammon Move Constraints.....</u>	<u>10</u>
 Bridge	11
<u>Bridge Codeless</u>	<u>11</u>
 Vecscript Code Listings	13
<u>Chess</u>	<u>13</u>
<u>Scramble</u>	<u>16</u>
<u>Backgammon</u>	<u>23</u>
<u>Bridge</u>	<u>25</u>

Overview

Introduction

[[Home](#)]

Vecset is used to create multiplayer board games, as well as animated games. You can log on to Vecset.net and play these games with other Vecset users. Vecset games are coded in a built-in scripting language called Vecscript, and then uploaded to the Vecset web site. Non-programmers can create drag-and-drop games, and programmers can add functionality to these games.

Drag-and-drop board games are constructed out of 5 basic components: 1) Card, such as a playing card or chess piece; 2) Card-stack, a stack of Card objects; 3) Board-grid, such as a chess or Go board; 4) Rack-grid, a collection of Card-stack objects arranged in a row (or column); and 5) Table-grid, a more flexible version of a Board-grid object (not just a plain grid), such as a Monopoly board.

Board Games Overview

This user guide describes using Vecset to make multiplayer Scramble, backgammon and bridge. Non-programmers are restricted to No-Code (codeless programming) drag-and-drop type games. Code 1 (Automation) is the easiest type of game programming, which involves automating various aspects of the game-player environment and game-playing process. Code 2 (Move Constraints), or checking for illegal moves, is a little harder. Code 3 (Auto Move), playing against the computer, is the most challenging type of game programming.

Game Coding

- **No-Code:** Codeless Programming utilizes 5 basic components: 1) Card (a collection of one or more labels and bitmaps); 2) Card-stack, a stack of Card objects; 3) Board-grid, a 2-dimensional array of Card-stack objects; 4) Rack-grid, a one-dimensional array of Card-stack objects; 5) Table-grid, similar to a Board-grid, but column widths and row heights may vary, and adjacent cells may be merged into a single cell.
- **Code 1:** Automation Programming, which is accomplished with custom Vecscript code that relieves some of the tedium of pure drag-and-drop games, overriding one or more default event handlers.
- **Code 2:** Move Constraints, or checking for illegal moves, are accomplished with custom Vecscript code which determines whether or not a given rule is broken.
- **Code 3:** Auto Move Programming, enabling the user to play against the computer, requires 4 game-specific classes: 1) Game-state, a mathematical model of the current game state; 2) Move, a mathematical model of the current move; 3) Move-generator, which generates the optimum move; and 4) Move-implementer, which carries out the move generated by Move-generator.

Board Game Codes

No-Code: Codeless Programming

Each of the 3 sample games covered here (Scramble, backgammon, and bridge) is based on the same basic set of 5 generic drag-and-drop components. These components enable the game designer to implement simple drag-and-drop versions of almost any board and card game without coding, hence the term "codeless programming." Each "codeless" game implementation contains no game-specific code. That way, even a game designer who is a non-programmer is capable of creating almost any board game imaginable.

Drag-and-Drop Components

Here are the 5 basic drag-and-drop components, the Card component, which contains the actual text and/or graphic(s) to be dragged, and its 4 container components:

- **Card:** This component corresponds to a playing card, chess piece, Scramble letter tile, Monopoly player piece, etc. It is always contained in a Card-stack object.
- **Card-stack:** A stack of Card objects. Individual Card objects, or entire Card-stack objects, can be dragged and dropped to other Card-stack objects.
- **Board-grid:** A 2-dimensional array of Card-stack objects. Each element of the array has the same width and height.
- **Rack-grid:** A one-dimensional array of Card-stack objects, oriented horizontally or vertically.
- **Table-grid:** A 2-dimensional array of Card-stack objects, in which the column widths and row heights may vary. Also, adjacent array elements may be merged into one table cell containing a single Card-stack object. This component would be a good candidate for a Monopoly board.

Tri-State Design Process

During the game design process, the game designer is always in one of the following 3 states: **design-time**, in which components are dropped on a form (game window) and their properties set; **run-time**, in which the game program is executed (what the end-user sees); and **live-design-time**, which is similar to run-time, except that the "freeze" command has not yet been invoked. The live-design-time state allows the game designer to drag Card objects to where they will be located at run-time (when the end-user starts the game). When the game designer is satisfied with the layout of the game, the freeze command is invoked, saving the current game setup in a form file with a .XML extension.

Code 1: Automation

Since pure drag-and-drop games can at times be a bit tedious for the players, it is often desirable to override one or more event handlers with custom Vecscript code. Here is a partial list of features that can be accomplished this way. Features that are relatively difficult to implement are in *italics*.

Scramble

- Fill rack
- Display no. of letters left
- Add score to total
- Prevent moving letters already down
- *Calculate score*
- *Click, not drag*

Bridge

- Deal hand
- Move hand to dummy
- Calculate winner of trick
- Calculate who is declarer
- Calculate who is dealer
- Count NS/WE tricks
- Add score to total
- *Calculate score*

Backgammon

- Roll dice
- Click, not drag
- *Detect gammon/backgammon*
- Keep score

Code 2: Move Constraints

One limitation of codeless versions of Scramble and other games is the lack of facility for preventing illegal moves. A move constraint is a piece of code that determines

whether or not the corresponding game rule is broken. Move constraints often work in conjunction with the game-state object, which encapsulates the current game-state.

Move constraints which are limited to preventing the dragging of Card or Card-stack objects from one container object to another can be accomplished without coding. This is due to the fact that every grid and Card-stack object has a property called `drop-list`, which is a list of objects that may accept drag-and-drop operations originating with the property owner.

Scramble

- Next Button
 - Word is connected
 - Word is contiguous
 - Word is in dictionary
 - Interchanged blank is used on same turn
- Double-Click on Board
 - Interchange blank if valid
- Drag from Board
 - Prevent dragging letters not in current turn
- Drop on Bag
 - If 3 of a kind, then accept
 - Else if user is scrambling, then accept

Code 3: Auto Move

Playing against the computer requires 4 game-specific classes: Game-state, Move, Move-generator, and Move-implementer.

Game State

A simplified, mathematical model of the current game state. For example, a chess board (the game-state of chess) consists of an 8 x 8 matrix of integers. Zero represents an empty square, 1 = a white pawn, -1 = a black pawn, 2 = a white knight, etc.

Move

A simplified, mathematical model of the current move. For example, a chess move consists of 4 integers: the source row/column and the destination row/column.

Move Generator

Generates the optimum move.

Move Implementer

Carries out the move generated by Move-generator.

Scramble

Scramble Codeless

Scramble is a board game in which players form words on a 15 x 15 grid, crossword-style, using the 7 letter tiles on their racks. You can use Vecset to make a purely drag-and-drop version of Scramble without doing any coding.

Design-Time

- **board:** Board-grid object, 15 x 15.
- **rack:** Rack-grid object of length 7. Boolean property local is set to true, meaning each player has own copy.
- **bag:** Card-stack object containing 100 letter tiles (Card objects).
- **table-top:** Card-stack object used for Scrambling. Boolean property local is set to true.
- **score-grid:** Editable-grid object, 6 rows by 2 columns. An Editable-grid is similar to a StringGrid object in Delphi, except that it can be edited at run-time (and at live-design-time, of course).

Live-Design-Time

The following form is for design purposes only and is hidden at run-time:

- **premium-card:** Card object containing one Text-box object, indicating the type of premium board square.
- **premium-grid:** Rack-grid object of length 4. Game designer copies and pastes premium-card into each cell, switches to design mode, and edits Text-box object of each cell and color of each cell. Boolean property infinite is set to true, so that dragging from any cell simply copies the single card in the cell and does not remove it. After creating and editing the premium-grid, the game designer copies and pastes it to the main Scramble form.
- **letter-card:** Card object containing 2 Text-box objects, a larger one for the letter and a smaller one for the point count.
- **alpha-grid:** Board-grid object, 6 rows by 5 columns. Game designer copies and pastes letter-card into each cell, switches to design mode, and edits each Text-box object of each cell. Boolean property infinite is set to true.
- **bag-grid:** Board-grid object, 10 rows by 10 columns. Game designer drags letter tiles from alpha-grid object.

- **bag:** Card-stack object. Game designer drags all letter tiles from bag-grid object. Boolean property random is set to true, meaning that the Card object obtained by dragging from bag is selected at random. After filling bag, the game designer copies and pastes it to the main Scramble form.
- **board:** The premium squares are dragged from the premium-grid object to the board object. Afterwards the premium-grid object may be deleted, since a copy exists on the Scramble design form.
- **score-grid:** The game designer edits all cells in the first column. Entries \$P1 - \$P4 are replaced by the names of up to 4 players at run-time. The name of the current player is highlighted. "Current" is short for Current Word Score. Anyone who uses a blank (or both blanks) must type in the corresponding letter(s).

Run-Time

- **Start of Game:** Player names are displayed. Player who goes first is selected at random and corresponding name is highlighted. The rack is initially empty and must be filled by dragging letter tiles from the bag.
- **Making a Word:** Player drags letters from rack to board.
- **Scrambling:** Player drags letters from rack to table-top, refills rack by dragging letters from bag to rack, then drags contents of table-top to bag.
- **Score Calculation:** Player calculates score manually, enters it into "Current" row of score-grid, adds it to his/her score total and replaces old total with new total.
- **End of Turn:** Player clicks on Next button, whose on-click event handler is pre-configured to call the next-player method of the game-server object.
- **Challenging:** Any player may challenge by checking the Challenge check box. Whoever has a dictionary looks up the word and reports on word validity in the chat window. If it's not a word, player who made the invalid word must click on Back button (another button with a pre-configured event handler), and take back the invalid word.
- **Filling Rack:** If no one challenges, player refills rack by dragging letters from bag to rack.

Scramble

Cameron	48
Mike	52
Current	
Blank	

L₁ N₁ O₁ E₁ U₁ S₁ E₁

 Challenge

Chat

Scramble Design

A ₁	A ₁	A ₁	A ₁	A ₁	A ₁	A ₁	A ₁	A ₁	B ₃
B ₃	C ₃	C ₃	D ₂	D ₂	D ₂	D ₂	E ₁	E ₁	E ₁
E ₁	E ₁	E ₁	E ₁	E ₁	E ₁	E ₁	E ₁	E ₁	F ₄
F ₄	G ₂	G ₂	G ₂	H ₄	H ₄	I ₁	I ₁	I ₁	I ₁
I ₁	I ₁	I ₁	I ₁	I ₁	J ₈	K ₅	L ₁	L ₁	L ₁
L ₁	M ₃	M ₃	N ₁	N ₁	N ₁	N ₁	N ₁	N ₁	O ₁
O ₁	O ₁	O ₁	O ₁	O ₁	O ₁	O ₁	P ₃	P ₃	Q ₁₀
R ₁	R ₁	R ₁	R ₁	R ₁	R ₁	S ₁	S ₁	S ₁	S ₁
T ₁	T ₁	T ₁	T ₁	T ₁	T ₁	U ₁	U ₁	U ₁	U ₁
V ₄	V ₄	W ₄	W ₄	X ₈	Y ₄	Y ₄	Z ₁₀		

A ₁	B ₃	C ₃	D ₂	E ₁
F ₄	G ₂	H ₄	I ₁	J ₈
K ₅	L ₁	M ₃	N ₁	O ₁
P ₃	Q ₁₀	R ₁	S ₁	T ₁
U ₁	V ₄	W ₄	X ₈	Y ₄
Z ₁₀				

Scramble Automation

By writing some Vecscript code, you can create a Scramble game which automates various aspects of the game, such as displaying a label indicating how many letters are left in the bag.

Feature List

The following 4 features are relatively easy to implement, by adding code to the appropriate event handlers:

1. Display no. of letters in bag
2. Add current word score to total
3. Prevent dragging of board letters already down
4. Fill rack at start of turn

Feature #3 requires that a Scramble-specific game-state object be present. This object has properties for the board, bag, each player's rack, etc. The board property is a 15 x 15 array of integers, where 0 represents an empty square, a positive value represents letters from previous turns, and a negative value represents letters from the current turn.

Scramble Move Constraints

The following is a list of 8 move constraints, each of which corresponds to its own rule governing the legality of Scramble moves. Four of these constraints affect the outcome of the user clicking the "Next" button: only if all 4 are true does the next-player method of the game-server object get called. Two more constraints govern mouse operations (double-clicking and dragging) on the board, and the final 2 constraints determine whether or not the user may drag the table-top Card-stack to the bag Card-stack.

Next Button

1. At least one letter placed on board in current turn must be adjacent to a letter placed on board in a previous turn. If this is the first turn, at least one letter must cover the center square.
2. If the word is across, all letters must be in the same row. If the word is down, all letters must be in the same column. There can be no empty squares in the middle of the word.
3. If challenging is disabled, all newly formed words, including those perpendicular to the main word, if any, must be in dictionary.

4. If interchange-blanks enabled, and use-on-same-turn enabled, then the no. of blanks placed on board must be greater than or equal to the no. of interchanged blanks picked up in current turn.

Double-Click on Board

If interchange-blanks enabled, and clicked board square is a blank, and value of blank exists on rack, then interchange the blank.

Drag from Board

Disable drag if letter being dragged was not placed on board in current turn (see Feature 3 in [Scramble Automation](#)). No code is necessary to prevent dragging of letter/word premium Card objects, since their draggable property is false. Also, the user cannot drag a letter tile on top of another letter tile on the board, since the coverable property of every letter tile is false.

Drop on Bag

The table-top Card-stack is the only object that can be dragged to the bag. The grabbable property of the table-top is true, meaning that the user can "grab" the entire table-top by dragging a small semicircular "handle" jutting out of the left border of the table-top.

1. If three-of-a-kind or four-of-a-kind enabled, and all letters in table-top, if included with letters on rack, are part of a 3-of-a-kind (or 4-of-a-kind), then accept drop. Otherwise do next rule.
2. Assume the user is Scrambling. Prompt the user for confirmation: if yes then accept drop and end user's turn. If no then cancel drop.

Scramble Auto Move

In order to implement playing Scramble against the computer, the following 4 game-specific classes are required: Game-state, Move, Move-generator, and Move-implementer.

Backgammon

Backgammon Codeless

The object of the game is to bring all of your game pieces safely home, while trying to avoid being sent to the "bar" by your opponent. Each turn you roll 2 dice, determining how many spaces to move your game piece(s).

Design-Time

- **board:** Board-grid object, 13 columns by 12 rows. Grid width property is set to zero pixels, Boolean property transparent is set to true. Horizontal offset (distance between adjacent cards) is set to one quarter of the diameter of a

white/black disk. Horizontal alignment property of each cell is set to "centered".

- **board-container:** Layer-book object with 2 layers. Bottom layer contains a bitmap of triangular, alternating light/dark "points". Top layer contains board object.
- **white-stack:** Card-stack object containing one Card object (a white, circular disk). Boolean property infinite is set to true.
- **black-stack:** Card-stack object containing one Card object (a black, circular disk). Boolean property infinite is set to true.
- **score-grid:** Editable-grid object, 2 columns by 2 rows. An Editable-grid is similar to a StringGrid object in Delphi, except that it can be edited at run-time. Boolean property persistent is set to true, meaning that the contents of this grid will survive a call of the new-game method of the game-server object.
- **dice:** Rack-grid object of length 2. Both cells contain a Card-stack object which consists of a stack of 6 bitmap-containing Card objects. Boolean property random of both Card-stack objects is set to true.
- **double-cube:** Card-stack object is a stack of six string-containing Card objects.
- **undo-double:** Card-stack object.

Run-Time

At run-time, the player clicks on both dice to "roll" them, and then moves his or her men, dragging them from the source grid squares to the destination grid squares. The player may double by clicking the double cube, which automatically moves the top card in the double cube Card-stack to the undo-double Card-stack. To undo a double operation, the player simply clicks on the undo-double Card-stack.

Backgammon Automation

Feature List

The following 2 features are relatively easy to implement, by adding code to the appropriate event handlers:

1. Roll dice at beginning of every turn
2. Update player scores at end of every game

Backgammon Move Constraints

The following is a list of 8 move constraints, each of which corresponds to its own rule governing the legality of backgammon moves. Six of these constraints have to do with dragging and dropping pieces on the board, and the other 2 constraints

affect the outcome of the user clicking the "Next" button: only if none of them are violated does the next-player method of the game-server object get called.

Drag from Board

1. Drag is enabled only if piece being dragged belongs to current player (exception: if current point contains exactly one piece of current player, and piece being dragged is the only piece of opposing player on this point, then enable drag).
2. Drag is disabled if piece being dragged is not on the bar and at least one piece of current player is on the bar.

Drop on Board

1. Disable drop if destination point contains more than one piece belonging to opposing player.
2. Let x = value of distance moved. If $x = 0$ then enable drop. Otherwise, disable drop if x not found in list of outstanding move values. In case of doubles, this list initially contains 4 integers (otherwise only 2 integers).
3. Disable drop if destination point is the home point and at least one piece belonging to current player is not in position to bear off.
4. Disable drop if piece belongs to opposing player and destination point is not the bar.

Next Button

1. Enable next button if list of outstanding move values is empty.
2. Otherwise, disable next button if the following is true: for all outstanding move values, if any move value is equal to at least one move value in the list of all possible moves for current player.

Bridge

Bridge Codeless

One of the most popular and challenging card games. The object of the game is to take as many "tricks" as possible. Each hand, you and your teammate bid against the opposing 2-player team to decide how many tricks are to be taken by the top bidding team. After the bidding is over, each player plays a card and the highest card takes the trick, and play continues until all cards have been played.

Design-Time

- **hand:** Rack-grid object of length 4 (no. of suits). Boolean property local is set to true, meaning each player has own copy. Orientation is vertical, not

horizontal, with horizontal offset (distance between adjacent cards) equal to 150 percent of the width of an upper-case letter.

- **dummy:** Rack-grid object of length 4 (no. of suits). Boolean property local is set to false, meaning only one copy exists (all players share the same copy). Orientation is vertical, not horizontal, with horizontal offset (distance between adjacent cards) equal to 150 percent of the width of an upper-case letter.
- **bidding-grid:** Table-grid object, 3 columns, as many rows as will fit in its space. Column headings: Player Name, Bid Level, Suit. The current player name (North, South, West, or East) is displayed in the first column, by entering \$PO in that column. The 2nd and 3rd columns are each associated with a combo box, enabling the user to select desired bid level and suit without having to type it in.
- **dummy-container:** A tabbed notebook object with 3 tabs, one for the dummy, one for the bidding-grid, and one for the score-grid.
- **score-grid:** Editable-grid object, 2 columns, as many rows as will fit in its space. An Editable-grid is similar to a StringGrid object in Delphi, except that it can be edited at run-time (and at live-design-time, of course). Persistent property is set to true, meaning that the contents of this grid will survive a call of the new-game method of the game-server object.
- **deck:** Std-deck object (a standard 52-card deck of playing cards).
- **trick:** Card-stack object, with horizontal offset (distance between adjacent cards) equal to 150 percent of the width of an upper-case letter.

Run-Time

- **Dealing:** All 4 players must deal 13 cards to themselves, by dragging cards from the deck to their hands.
- **Bidding:** Name of current player (North, South, West, or East) is displayed in 1st column of bidding-grid, when clicked on. Current player selects bid level and suit using combo boxes in 2nd and 3rd columns of bidding-grid. At end of bidding, declarer's partner drags each Card-stack (suit) in his/her hand to dummy.
- **Playing a Card:** Player clicks on desired card, sending it to the trick Card-stack, and then clicks on "Next Player" button (or presses Enter).
- **Taking the trick:** Trick-taker clicks on "handle" of trick Card-stack, sending it to the deck Card-stack. Trick-taker then clicks on appropriate trick-counter Counter-object (there are 2 of these: North/South and West/East).
- **End of hand:** Score-keeper calculates score manually, enters it into score-grid, and clicks on the "New Hand" button, which is like a "New Game" button except that the score Table-grid is not re-initialized.

Vecscript Code Listings

Chess

The source code listing below is a partial implementation of a computer chess game.

Source Code

```
(class (public) Chess-form extends Form
  var (auto) (
    Game-server game-server;
    Board-grid board-grid;
    ...
  )
  var (private) (
    Game-state game-state;
    ...
  )
  (proc (auto) board-grid_change (Event e)
    do (
      : game-state (set-board (: e row) (: e col) (: e intval));
    )
  )
  (proc (auto) game-server_auto-move (Event e)
    var (
      Move-generator mg (new Move-generator game-state);
      Move-implementer mi (new Move-implementer game-state);
    )
    do (
      : mi do-move (: mg get-move);
    )
  )
  ...
)
(class Game-state
  var (private) (
    array 2 int board (new int 8 8);
    boolean white;
    boolean check;
    boolean en-passant;
    int en-passant-col;
    Player-state white-player;
    Player-state black-player;
  )
  (func (public) int get-board (int row; int col)
    do (
      return (board row col);
    )
  )
  (proc (public) set-board (int row; int col; int val)
    do (
      = (board row col) val;
    )
  )
  (func (public) Player-state get-player (boolean white)
```



```

    ))
    (func (public) int get-dest-col do (
      return dest-col;
    ))
    ...
  )
(class Move-generator
  var (private) (
    Game-state game-state;
  )
  (cons Move-generator (Game-state gs)
    do (
      = game-state gs;
    )
  )
  (func (public) Move get-move
    var (private) (
      Move best-move;
    )
    do (
      { calculate optimum move }
      ...
      return best-move;
    )
  )
)
(class Move-implementer
  var (private) (
    Game-state game-state;
  )
  (cons Move-implementer (Game-state gs)
    do (
      = game-state gs;
    )
  )
  (proc (public) do-move (Move move)
    do (
      { handle special cases... }
      if (== move nil) then (
        { check mate }
      )
      elseif (move is-castling) then (
        { computer is castling }
      )
      elseif (move is-en-passant-capture) then (
        { must remove captured pawn }
      )
      elseif (move is-queened-pawn) then (
        { promote pawn to queen }
      )
      else (
        { normal case }
        : board-grid (auto-drop
          (: move get-dest-row)(: move get-dest-col)
          (: board-grid (auto-drag
            (: move get-source-row)(: move get-source-col)))
        );
      );
    );
  );
)

```

```
)  
)  
)
```

Scramble

Scramble Codeless

The following code is generated automatically by Vecscript:

```
import (Hidden-form)  
  
(class (public) Scrab-form extends Form  
  var (auto) (  
    Game-server game-server;  
    Button next-btn;  
    Button back-btn;  
    Button new-game-btn;  
    Button exit-btn;  
    Board-grid board-grid;  
    Rack-grid rack-grid;  
    Card-stack bag;  
    Card-stack table-top;  
    Editable-grid score-grid;  
  )  
(proc (public static) main (String-list args)  
  var (  
    Scrab-form form1 (new Scrab-form);  
    Hidden-form form2 (new Hidden-form);  
  )  
)  
(proc (public) Scrab-form  
  do (  
    : game-server init-game;  
    show;  
  )  
)  
(proc (auto) next-btn_click (Event e)  
  do (  
    : game-server next-player;  
  )  
)  
(proc (auto) back-btn_click (Event e)  
  do (  
    : game-server previous-player;  
  )  
)  
(proc (auto) new-game-btn_click (Event e)  
  do (  
    : game-server new-game;  
  )  
)  
(proc (auto) exit-btn_click (Event e)  
  do (  
    : game-server quit-player;  
  )  
)
```

```
)  
)
```

This class belongs in a separate source file:

```
(class (public) Hidden-form extends Form  
  var (auto) (  
    Card premium-card;  
    Rack-grid premium-grid;  
    Card letter-card;  
    Board-grid alpha-grid;  
    Board-grid bag-grid;  
    Card-stack bag;  
  )  
)
```

Scramble Automation

Feature No. 1: Display no. of letters in bag

```
(proc (auto) bag_change (Event e)  
  do (  
    : bag-count-lbl (set-caption (: bag get-count));  
  )  
)
```

Feature No. 2: Add current word score to total

```
(proc (auto) next-btn_click (Event e)  
  var (int i)  
  do (  
    = i (- (: game-server get-player-no) 1); \ get current player no.  
    : score-grid (set-cell i 1 (+  
      (str-to-int (: score-grid (get-cell i 1)))  
      (str-to-int (: score-grid (get-cell 4 1)))  
    ));  
    : game-server next-player;  
  )  
)
```

Feature No. 3: Prevent dragging of board letters already down

```
(proc (auto) board-grid_change (Event e)  
  do (  
    : game-state (set-board (: e row) (: e col)  
      (- (abs (: board-grid (cells (: e row) (: e col))  
        get-top value))));  
  )  
)  
  
(proc (auto) next-btn_click (Event e)  
  var (int i; int j; int m; int n)  
  do (  
    = m (- (: board-grid get-row-count) 1);  
    = n (- (: board-grid get-col-count) 1);  
    for i (0 m) do (  

```

```

        for j (0 n) do (
          : game-state (set-board i j
            (abs (get-board i j)));
        );
      );
      : game-server next-player;
    )
  )
)

(proc (auto) board-grid_drag (Event e)
  do (
    = (: e can-drag) (<
      (: game-state (get-board (: e row) (: e col))) 0);
  )
)

```

Feature No. 4: Fill rack at start of turn

```

(proc (event) next-btn_click (Event e)
  do (
    fill-rack;
    : game-server next-player;
  )
)

{ fill all racks at start of game }

(proc init-game
  var (int i)
  do (
    : game-server first-player;
    for i (1 (: game-server get-player-count)) do (
      fill-rack;
      : game-server next-player;
    );
    { rest of init-game... }
  )
)

(proc fill-rack
  var (int i)
  do (
    for i (0 (- (: rack-grid count) 1)) do (
      if (<= (: bag count) 0) then (
        break;
      );
      if (== (: rack-grid (cells i) count) 0) then (
        : rack (cells i) push (: bag pop);
      );
    );
  )
)

```

Scramble Move Constraints

```

var (private) (
  boolean challenging;

```

```

boolean interchange-blanks;
boolean use-on-same-turn;
boolean three-of-a-kind;
boolean four-of-a-kind;
)
(proc (auto) next-btn_click (Event e)
  var (String msg)
  do (
    = msg '';
    if (== (: game-state (get-board 7 7)) 0) then (
      = msg 'Center square not covered';
    )
    elseif (not is-word-connected) then (
      = msg 'Word not adjacent to other words';
    )
    elseif (not is-word-contiguous) then (
      = msg 'Letters in word not contiguous';
    )
    elseif (not is-challenging) then (
      = msg get-words-not-in-dict;
      if (<> msg '') then (
        = msg (+ 'Word(s) not in dictionary: ' msg);
      );
    )
    elseif (and is-interchangeable-blanks is-use-on-same-turn
      (< get-blank-count-this-turn get-interchg-blank-count))
    then (
      = msg 'Not enough blank(s) used';
    );

    if (== msg '') then (
      : game-server next-player;
    )
    else (
      : show-err-msg msg;
    );
  )
)
(proc (auto) board_dbl_click (Event e)
  do (
    if (and is-interchangeable-blanks
      (is-blank-at (: e row) (: e col))
      (has-letter (get-blank-val-at (: e row) (: e col))))
    then (
      do-blank-interchange (: e row) (: e col);
    );
  )
)
(proc (auto) bag_drop (Event e)
  do (
    if (and is-three-of-a-kind is-part-of-3-of-a-kind) then (
      : e (set-accept true);
    )
    elseif (and is-four-of-a-kind is-part-of-4-of-a-kind) then (
      : e (set-accept true);
    )
    elseif (== (message-dlg 'Do you wish to Scramble?')

```

```

        mt-confirmation (mb-yes mb-no)) mr-yes) then (
          : e (set-accept true);
          : game-server next-player;
        )
      else (
        : e (set-accept false);
      );
    )
  )
)

{ Helper Methods }

(func boolean is-challenging do (return challenging;))
(func boolean is-interchangeable-blanks
  do (return interchangeable-blanks;))
(func boolean is-use-on-same-turn do (return use-on-same-turn;))
(func boolean is-three-of-a-kind do (return three-of-a-kind;))
(func boolean is-four-of-a-kind do (return four-of-a-kind;))

(func boolean is-word-connected do ( ... ))
(func boolean is-word-contiguous do ( ... ))
(func String get-words-not-in-dict do ( ... ))
(func int get-blank-count-this-turn do ( ... ))
(func int get-interchg-blank-count do ( ... ))
(func boolean is-blank-at (int row; int col) do ( ... ))
(func boolean has-letter (int letter) do ( ... ))
(func int get-blank-val-at (int row; int col) do ( ... ))
(proc do-blank-interchange (int row; int col) do ( ... ))
(func boolean is-part-of-3-of-a-kind do ( ... ))
(func boolean is-part-of-4-of-a-kind do ( ... ))
(proc show-err-msg (String msg) do ( ... ))

```

Scramble Auto Move

The (abbreviated) source code listing below shows how to implement Scramble using the on-change events and the 4 Auto Move classes.

On Change Events

```

(class (public) Scrab-form extends Form
  var (auto) (
    Game-server game-server;
    Board-grid board-grid;
    Rack-grid rack-grid;
    Card-stack bag;
    Card-stack table-top;
    Editable-grid score-grid;
  )
  var (private) (
    Game-state game-state;
    ...
  )
  (proc (auto) board-grid_change (Event e)
    do (
      : game-state (set-board (: e row) (: e col) (: e intval));
    )
  )
)

```

```

    )
  )
  (proc (auto) rack-grid_change (Event e)
    do (
      : game-state curr-player (set-rack (: e col) (: e intval));
    )
  )
  (proc (auto) game-server_auto-move (Event e)
    var (
      Move-generator mg (new Move-generator game-state);
      Move-implementer mi (new Move-implementer game-state);
    )
    do (
      : mi do-move (: mg get-move);
    )
  )
  ...
)

```

Game State

```

(class Game-state
  var (private) (
    array 2 int board (new int 15 15);
    Player-state curr-player;
    list Player-state player-list;
  )
  (func (public) int get-board (int row; int col)
    do (
      return (board row col);
    )
  )
  (proc (public) set-board (int row; int col; int val)
    do (
      = (board row col) val;
    )
  )
)
(class Player-state
  var (private) (
    array int rack (new int 7);
  )
  (func (public) int get-rack (int col)
    do (
      return (rack col);
    )
  )
  (proc (public) set-rack (int col; int val)
    do (
      = (rack col) val;
    )
  )
)
)

```

Move

```

(class Move
  var (private) (
    array int mv-word (new int 7);
    boolean across;
    int row;
    int col;
    int blank-pos-1;
    int blank-pos-2;
  )
  (func (public) int get-mv-word (int idx)
    do (
      return (mv-word idx);
    )
  )
  (proc (public) set-mv-word (int idx; int val)
    do (
      = (mv-word idx) val;
    )
  )
  (func (public) int get-row do (
    return row;
  ))
  (func (public) int get-col do (
    return col;
  ))
  (func (public) int get-board-row (int i) do (
    { calculate board row of i-th letter }
  ))
  (func (public) int get-board-col (int i) do (
    { calculate board col of i-th letter }
  ))
  (func (public) int get-rack-col (int i) do (
    { calculate rack col of i-th letter }
  ))
  ...
)

```

Move Generator

```

(class Move-generator
  var (private) (
    Game-state game-state;
  )
  (cons Move-generator (Game-state gs)
    do (
      = game-state gs;
    )
  )
  (func (public) Move get-move
    var (private) (
      Move best-move;
    )
    do (
      { calculate optimum move }
      ...
      return best-move;
    )
  )
)

```

```
)  
)
```

Move Implementer

```
(class Move-implementer  
  var (private) (  
    Game-state game-state;  
  )  
(cons Move-implementer (Game-state gs)  
  do (  
    = game-state gs;  
  )  
)  
(proc (public) do-move (Move move)  
  var (  
    int i;  
    int ltr-val;  
  )  
  do (  
    { handle special cases... }  
    if (== (: move get-row) -1) then (  
      { player is scrabbling or passing }  
    )  
    else (  
      { normal case }  
      for i (0 6) do (  
        if (or (== i (: move get-blank-pos-1))  
              (== i (: move get-blank-pos-2))) then (  
          = ltr-val 27;  
          \ store blank info...  
        )  
        else (  
          = ltr-val (: move (get-mv-word i));  
        );  
        if (== ltr-val 0) then (  
          break;  
        );  
        board-grid (auto-drop  
          (: move (get-board-row i)) \ board row of i-th letter  
          (: move (get-board-col i)) \ board col of i-th letter  
          (: rack-grid (auto-drag (: move (get-rack-col i))))  
        );  
      );  
    );  
  );  
)  
)  
)
```

Backgammon

Backgammon Codeless

The following code is generated automatically by Vecscript:

```
(class (public) Bg-form extends Form
```

```

var (auto) (
  Game-server game-server;
  Button next-btn;
  Button back-btn;
  Button new-game-btn;
  Button exit-btn;
  Board-grid board-grid;
  Layer-book board-container;
  Bitmap board-bitmap;
  Card-stack white-stack;
  Card-stack black-stack;
  Editable-grid score-grid;
  Rack-grid dice;
  Card-stack double-cube;
  Card-stack undo-double;
)
(proc (public static) main (String-list args)
  var (
    Bg-form form1 (new Bg-form);
  )
)
(proc (public) Bg-form
  do (
    : game-server init-game;
    show;
  )
)
(proc (auto) next-btn_click (Event e)
  do (
    : game-server next-player;
  )
)
(proc (auto) back-btn_click (Event e)
  do (
    : game-server previous-player;
  )
)
(proc (auto) new-game-btn_click (Event e)
  do (
    : game-server new-game;
  )
)
(proc (auto) exit-btn_click (Event e)
  do (
    : game-server quit-player;
  )
)
)
)

```

Backgammon Automation

Feature No. 1: Roll dice at beginning of every turn

```

(proc (auto) next-btn_click (Event e)
  do (
    : game-server next-player;
    roll-dice;
  )
)

```

```

    )
  )
  (proc roll-dice
    do (
      : dice-grid init;
    )
  )
)

```

Feature No. 2: Update player scores at end of every game

```

(proc (auto) new-game-btn_click (Event e)
  do (
    update-score-grid;
    : game-server new-game;
  )
)
{
  white-cb: check box with caption "White Wins"
  win-tyt-grp: radio group with 3 radio buttons:
  - Normal
  - Gammon
  - Backgammon
}
(proc update-score-grid
  var (
    int score-row;
    int score;
  )
  do (
    = score-row (? (: white-cb checked) 0 1);
    = score (* (+ 1 (: win-tyt-grp item-index))
      (str-to-int (: double-cube get-top
        text-boxes get-obj get-text)))
    );
    : score-grid (set-cell score-row 1 (+ score
      (str-to-int (: score-grid (get-cell score-row 1))))
    );
  )
)
)

```

Bridge

Bridge Codeless

The following code is generated automatically by Vecscript:

```

(class (public) Bridge-form extends Form
  var (auto) (
    Game-server game-server;
    Button next-btn;
    Button back-btn;
    Button new-hand-btn;
    Button exit-btn;
    Rack-grid hand;
    Rack-grid dummy;
  )
)

```

```

Card-stack trick;
Std-deck deck;
Table-grid bidding-grid;
Editable-grid score-grid;
Tabbed-notebook dummy-container;
)
(proc (public static) main (String-list args)
  var (
    Bridge-form form1 (new Bridge-form);
  )
)
(proc (public) Bridge-form
  do (
    : game-server init-game;
    show;
  )
)
(proc (auto) next-btn_click (Event e)
  do (
    : game-server next-player;
  )
)
(proc (auto) back-btn_click (Event e)
  do (
    : game-server previous-player;
  )
)
(proc (auto) new-hand-btn_click (Event e)
  do (
    : game-server new-game;
  )
)
(proc (auto) exit-btn_click (Event e)
  do (
    : game-server quit-player;
  )
)
)
)

```