```
package main

import (
	"fmt"
	"os"
	"strconv"
)

type Board struct {
	//each board has 2 fields, the position(x,y) and the size of checkerboard
	//each square contains some coins
	square [][]int
	//size will be SIZE
	size int
}

//the checkerboard create a size*size board
func InitializeCheckerBoard(SIZE int) *Board {
	board := make([][]int, SIZE)
	for i := 0; i < SIZE; i++ {
		board[i] = make([]int, SIZE)
	}
	return &Board{square: board, size: SIZE}
}

//Board.Contains() will check if certain indices belong to checkerboard, just
like InField()
func (board *Board) Contains(r, c int) bool {
	switch {
	case r < 0 || r > board.size:
		return false
	case c < 0 || c > board.size:
		return false
	default:
		return true
	}
}

//Topple() will check the number of coins and redistribute them if coins>4
//use a for loop to make it can not be toppled anymore
func (board *Board) Topple(r, c int) {
	//whild coins>=4,topple that square
	for board.square[r][c] >= 4 {
		//then give each InField-neighbor a coin in 4 directions:north south
east and west
		//left neightbor
		if board.Contains(r-1, c) {
			board.square[r-1][c] += 1
			board.square[r][c] -= 1
		} else if !board.Contains(r-1, c) { //even lies on boundary it will
still remove 1 coin,case-specified
			board.square[r][c] -= 1
		}
		//then do the same thing for other 3 neighbors
```

```
            if board.Contains(r+1, c) {
                  board.square[r+1][c] += 1
                  board.square[r][c] -= 1
            } else if !board.Contains(r+1, c) {
                  board.square[r][c] -= 1
            }
            if board.Contains(r, c-1) {
                  board.square[r][c-1] += 1
                  board.square[r][c] -= 1
            } else if !board.Contains(r, c-1) {
                  board.square[r][c] -= 1
            }
            if board.Contains(r, c+1) {
                  board.square[r][c+1] += 1
                  board.square[r][c] -= 1
            } else if !board.Contains(r, c+1) {
                  board.square[r][c] -= 1
            }
      }
}

//Sets the value of Cell(r,c)
func (board *Board) Set(r, c, value int) {
      //ensure valid square
      if board.Contains(r, c) {
            board.square[r][c] = value
      } else {
            fmt.Println("Error: not valid square")
            os.Exit(1)
      }
}

//returns the value of cell(r,c)
func (board *Board) Cell(r, c int) int {
      if !board.Contains(r, c) {
            fmt.Println("Error: not valid square")
            os.Exit(1)
      }
      return board.square[r][c]
}

//returns true if there are no cells with >=4 coins on them and false otherwise
func (board *Board) IsConverged() bool {
      //just loop over all cells and check their number of coins
      for i := 0; i < board.size; i++ {
            for j := 0; j < board.size; j++ {
                  if board.square[i][j] >= 4 {
                        return false
                  }
            }
      }
      return true
}
```

```
//returns the number of rows on the board
func (board *Board) NumRows() int {
	//board is size*size
	return board.size
}

//returns the number of colums on the board
func (board *Board) NumCols() int {
	return board.size
}

//repeatedly topple until we can't topple anymore to reach a stable
configuration
func (board *Board) EvolveToStable() {
	//if not stable
	if !board.IsConverged() {
		for i := 0; i < board.size; i++ {
			for j := 0; j < board.size; j++ {
				//topple()will run until <4 coins
				board.Topple(i, j)
			}
		}
	}
}

//now we draw the stable configuration
func (board *Board) DrawStableBoard() {
	//initialize the canvas
	pic := CreateNewCanvas(board.size, board.size)
	//we have four kinds of color
	black := MakeColor(0, 0, 0)       // 0
	gray1 := MakeColor(85, 85, 85)    // 1
	gray2 := MakeColor(170, 170, 170) // 2
	white := MakeColor(255, 255, 255) // 3
	//now loop over the whold board and fill the squares
	for i := 0; i < board.size; i++ {
		for j := 0; j < board.size; j++ {
			switch {
			case board.square[i][j] == 0:
				pic.SetFillColor(black)
			case board.square[i][j] == 1:
				pic.SetFillColor(gray1)
			case board.square[i][j] == 2:
				pic.SetFillColor(gray2)
			case board.square[i][j] == 3:
				pic.SetFillColor(white)
			}
			//after choosing color, we fill the square
			//1*1 square
			pic.ClearRect(i, j, i+1, j+1)
		}
	}
	pic.SaveToPNG("board.png")
}
```

```
func main() {

	//first check the number of arguments
	if len(os.Args) != 3 {
		fmt.Println("Error: SIZE PILE are needed")
		os.Exit(1)
	}
	//now pharse arguments

	SIZE, err := strconv.Atoi(os.Args[1])
	//positive integers
	if err != nil {
		fmt.Println("Error: SIZE must be integer")
		os.Exit(1)
	} else if SIZE <= 0 {
		fmt.Println("Error: SIZE must be positive")
		os.Exit(1)
	}

	//PILE is the number of coins
	PILE, err := strconv.Atoi(os.Args[2])
	//positive integers
	if err != nil {
		fmt.Println("Error: PILE must be integer")
		os.Exit(1)
	} else if PILE <= 0 {
		fmt.Println("Error: PILE must be positive")
		os.Exit(1)
	}

	//now initialize our sandpile
	board := InitializeCheckerBoard(SIZE)
	//pile of coins placed on the middle square
	board.Set(SIZE/2, SIZE/2, PILE)
	//after initialization, we evolve it
	board.EvolveToStable()
	//then we draw the final configuration
	board.DrawStableBoard()

}
```