Pesten

Izaak van Dongen

April 26, 2018

Contents

1 Introduction **1**

2 Rules **2**
- 2.1 Basic play . . . 2
- 2.2 Special cards . . . 2
- 2.3 Card sets . . . 3
- 2.4 Last (card), but not least . . . 3

3 Implementation **3**

Listings

1 UCard.pas . . . 3
2 UPack.pas . . . 4
3 UHand.pas . . . 6
4 UUIQuerier.pas . . . 9
5 UGame.pas . . . 10
6 UUI.pas . . . 13
7 UGameHandler.pas . . . 14

1 Introduction

Pesten is a classic Dutch card game, similar to Uno but played with playing cards. The name translates as something like ‘bothering’. The objective is to annoy your fellow players as much as possible. The suits function as colours, and various cards have special functions. There is also a fine tradition of introducing house rules. People can feel very strongly about these, and we’re still finding special cases that require and appeal to the van Dongen jury, so while this implementation aims to codify the van Dongen house rules, it makes no guarantee of absolute accuracy.

It is played with one to two packs of cards, but this is entirely variable depending on how many players there are.

2 Rules

2.1 Basic play

The basic functioning of the game is, as mentioned, very similar to Uno. There is a discard pile and a pickup pile. The top card of the discard pile determines the current player's allowable discards. A player is allowed to play any card of either the same suit or the same rank as the current card, or a joker, unless there is currently a special effect in play…

If a player is able to play, they must. However, if they can't, they pick up one card from the draw pile. If they are able to play this card, they may, and this card takes effect as normal.

2.2 Special cards

Some cards have effects, which generally apply to the next player. Due to this, sometimes players opposite each other form 'teams' as they never obstruct each other. This mode of play also allows everyone to win more often. All effects are listed here:

Card	Effect
Joker	This card has no suit so can be played on any card (unless this table specifies otherwise, see 2). The following player must take 5 cards, or play their own joker, which increases the count to 10 and moves to the next player. The player who ends up taking the cards may decide on the initial suit after the joker, but may not play. Play goes to the next player after the suit has been decided.
2	The following player must take 2 cards. If the following player has a 2, they may play it and then the total number of cards to be taken is 4, by the next player, and so on. A player may also 'escalate' by playing a 3 of the same rank. This increases the payload by three. A following player must then play a 4 of the same rank or a 3, and so on. Any cards played in this mode are exempt from their normal special effects. A joker cannot be played while a 2-stack is in play. The player who ends up taking the cards may not play, and play goes to the next player from them.
7	A 7 allows the player to take another turn. Their next card must fit on the seven, or they will have to pick up a card.
8	An 8 skips the next player. This action cannot be stopped by any card, as the next player simply doesn't get a turn, so cannot do anything like play their own 8.
10	A 10 means the player before the current player now has their turn, but play goes on as normal.
Jack	A jack lets the player choose the suit to go on with. The next player must play a jack, or a card of the declared suit, or a joker.
King	Changes the direction of play. The next turn goes the player originally 'before' the current player.

Any unmentioned cards are not special.

2.3 Card sets

A player may also play a ‘set’ of cards. A set of cards is either three or more of the same rank, or three or more adjacent cards of the same suit in ascending or descending order. For example, one might play 6 ♣, 6 ♢, 6 ♠, or 6 ♣, 5 ♣, 4 ♣. NB for the purpose of these sets, aces are considered to be both before 2 and after the King.

A set must contain at least three cards following the pattern. However, the play of two cards is permitted if they form a set with the top card on the discard pile, eg you can play 6 ♢, 6 ♠ if there is a 6 ♣ at the top of the discard pile.

If there are special effect cards, only the top card has its effect. This means that it’s a better idea to play 9 ♣, 8 ♣, 7 ♣ than the other way round, as this gives the player another turn.

If a card set is played while a 2-stack is building, only the last card of the set contributes to the number to be picked up.

2.4 Last (card), but not least

As soon as a player reaches their last card, they must tap the table twice and declare ‘last card’ with appropriate volume and enthusiasm. If they don’t, a two-card penalty is inflicted. If the player is playing a series of cards (using 7s, or sets) and they forget last card, they take back their last card and two penalty cards. They may not continue to play.

3 Implementation

```
{$MODE OBJFPC}

unit UCard;

interface

uses SysUtils;

const
    suits: array[0..3] of string = ('Spades', 'Clubs', 'Hearts', 'Diamonds');
    char_suits: array[0..3] of string = ('♠', '♣', '♡', '♢');
    ranks: array[0..12] of string =
                        ('Ace', 'Two', 'Three', 'Four', 'Five', 'Six', 'Seven',
                         'Eight', 'Nine', 'Ten', 'Jack', 'Queen', 'King');
    char_ranks: array[0..12] of char = ('A', '2', '3', '4', '5', '6', '7',
                                        '8', '9', 'T', 'J', 'Q', 'K');

type
    ECardError = class(Exception);

    TCard = class
        private
            Rank, Suit: Integer;
        public
            constructor Create(r, s: integer);
            function GetRank: integer;
            function GetSuit: integer;
            function GetScore: integer;
            function GetAltScore: integer;
            function GetName: string;
            function GetShortName: string;
```

```pascal
    end;

    TCardArray = array of TCard;
    TCardKeyFunc = function(card: TCard): integer;

function proper_mod(a, b: integer): integer;

implementation

{global functions}

function proper_mod(a, b: integer): integer;
begin
    proper_mod := a mod b;
    if proper_mod < 0 then
        proper_mod := proper_mod + b;
end;

{methods}

constructor TCard.Create(r, s: integer);
begin
    Rank := r;
    Suit := s;
end;

function TCard.GetRank: integer;
begin
    result := Rank;
end;

function TCard.GetSuit: integer;
begin
    result := Suit;
end;

function TCard.GetScore: integer;
begin
    result := Rank * 4 + Suit;
end;

function TCard.GetAltScore: integer;
begin
    result := Suit * 4 + Rank;
end;

function TCard.GetName: string;
begin
    result := Format('%s of %s', [ranks[Rank], suits[Suit]]);
end;

function TCard.GetShortName: string;
begin
    result := Format('%s%s', [char_ranks[Rank], char_suits[Suit]]);
end;

end.
```

Listing 1: UCard.pas

```pascal
{$MODE OBJFPC}

unit UPack;

interface
```

```pascal

uses SysUtils, UCard;

type
    TPack = class
        private
            cards: TCardArray;
            all_cards: TCardArray;
            bottom, ncards, num_packs: integer;
            procedure Populate;
        public
            constructor Create(n: integer);
            destructor Free;
            function GetSize: integer;
            procedure Shuffle;
            function Deal: TCard;
            procedure ReturnCard(card: TCard);
    end;

implementation

constructor TPack.Create(n: integer);
begin
    bottom := 0;
    ncards := 52 * n;
    num_packs := n;
    cards.setlength(ncards);
    Populate;
    Shuffle;
end;

destructor TPack.Free;
var
    i: integer;
begin
    for i := 0 to 51 do
        all_cards[i].free;
end;

function TPack.GetSize: integer;
begin
    result := ncards;
end;

procedure TPack.Populate;
var
    i, j: integer;
begin
    for j := 0 to num_packs - 1 do
        for i := 0 to 51 do begin
            cards[i] := TCard.create(i mod 13, i div 13);
            all_cards[i] := cards[i];
        end;
end;

procedure TPack.Shuffle;
var
    i, ind_a, ind_b: integer;
    temp: TCard;
begin
    for i := ncards - 1 downto 1 do begin
        ind_a := proper_mod(random(i) + bottom, cards.length);
        ind_b := proper_mod(i, cards.length);
        temp := cards[ind_b];
        cards[ind_b] := cards[ind_a];
        cards[ind_a] := temp;
    end;
end;

function TPack.Deal: TCard;
begin
    if ncards = 0 then
        raise ECardError.create('can''t deal card as pack is empty')
    else begin
        result := cards[proper_mod(bottom + ncards, cards.length)];
        dec(ncards);
    end;
end;

procedure TPack.ReturnCard(card: TCard);
begin
    if ncards = cards.length then
        raise ECardError.create('can''t return card as pack is full')
    else begin
        cards[bottom] := card;
        bottom := proper_mod(bottom - 1, cards.length);
        inc(ncards)
    end;
end;

initialization

begin
    randomize;
end;

end.
```

Listing 2: UPack.pas

```pascal
{$MODE OBJFPC}

unit UHand;

interface

uses SysUtils, UCard;

type
    TKeyArray = array of integer;

    type THand = class
        protected
            cards: TCardArray;
            size: integer;
            procedure Sort(cardbuf: TCardArray; keybuf, keys: TKeyArray; lower, upper:
    ↪ integer);
            procedure Merge(cardbuf: TCardArray; keybuf, keys: TKeyArray; lower, mid, upper:
    ↪  integer);
        public
            constructor Create;
            function GetSize: integer;
            function Display: string;
            procedure PushCard(card: TCard);
            procedure InsertCard(card: TCard; i: integer);
            function RemoveCard(i: integer): TCard;
            function PopCard: TCard;
            procedure ClearHand;
            function ViewCard(i: integer): TCard;
            function TopCard: TCard;
            procedure SwapCards(i, j: integer);
            procedure Sort(keyfunc: TCardKeyFunc);
            procedure SortByRank;
            procedure SortBySuit;
    end;

implementation

constructor THand.Create;
begin
    size := 0;
end;

function THand.GetSize: integer;
begin
    result := size;
end;

function THand.Display: string;
var
    i: integer;
begin
    result := 'Hand(';
    if size > 0 then
        result := result + cards[0].GetShortName;
    for i := 1 to size - 1 do
        result := result + ', ' + cards[i].GetShortName;
    result := result + ')';
end;

procedure THand.PushCard(card: TCard);
begin
    if size > cards.length - 1 then
        raise ECardError.create('can''t add card to hand as it is full');
    cards[size] := card;
    inc(size);
end;

function THand.PopCard: TCard;
begin
    if size = 0 then
        raise ECardError.create('can''t discard as hand is empty')
    else begin
        result := cards[size - 1];
        dec(size);
    end;
end;

procedure THand.ClearHand;
begin
    size := 0;
end;

procedure THand.InsertCard(card: TCard; i: integer);
var
    j: integer;
begin
    if size > cards.length - 1 then
        raise ECardError.create('can''t add card to hand as it is full');
    if (i < 0) or (i >= size) then
        raise ECardError.create('can''t add card, this is an invalid index');
    for j := size downto i + 1 do
        cards[j] := cards[j - 1];
    cards[i] := card;
    inc(size);
```

```pascal
end;

function THand.RemoveCard(i: integer): TCard;
begin
    if size = 0 then
        raise ECardError.create('can''t remove card, this hand is empty');
    if (i < 0) or (i >= size) then
        raise ECardError.create('can''t add card, this is an invalid index');
    result := cards[i];
    for i := i to size - 2 do
        cards[i] := cards[i + 1];
    dec(size);
end;

function THand.ViewCard(i: integer): TCard;
begin
    if (i >= size) or (i < 0) then
        raise ECardError.create('can''t view card outside of range')
    else
        result := cards[i];
end;

function THand.TopCard: TCard;
begin
    result := ViewCard(size - 1);
end;

procedure THand.SwapCards(i, j: integer);
var
    tmp_card: TCard;
begin
    if (i >= size) or (j >= size) then
        raise ECardError.create('can''t swap card outside of range')
    else
        tmp_card := cards[i];
        cards[i] := cards[j];
        cards[j] := tmp_card;
end;

procedure THand.Sort(cardbuf: TCardArray; keybuf, keys: TKeyArray; lower, upper: integer);
var
    mid: integer;
begin
    if upper - lower > 1 then begin
        mid := (lower + upper) div 2;
        Sort(cardbuf, keybuf, keys, lower, mid);
        Sort(cardbuf, keybuf, keys, mid, upper);
        Merge(cardbuf, keybuf, keys, lower, mid, upper);
    end;
end;

procedure THand.Merge(cardbuf: TCardArray; keybuf, keys: TKeyArray; lower, mid, upper:
    ↪ integer);
var
    i, j, k: integer;

begin
    i := lower;
    j := mid;
    k := 0;
    while (i < mid) and (j < upper) do
        if keys[i] <= keys[j] then begin
            keybuf[k] := keys[i];
            cardbuf[k] := cards[i];
            inc(i);
            inc(k);
        end else begin
            keybuf[k] := keys[j];
            cardbuf[k] := cards[j];
            inc(j);
            inc(k);
        end;

    for i := i to mid - 1 do begin
        keybuf[k] := keys[i];
        cardbuf[k] := cards[i];
        inc(k);
    end;

    for j := j to upper - 1 do begin
        keybuf[k] := keys[j];
        cardbuf[k] := cards[j];
        inc(k);
    end;

    for i := 0 to k do begin
        keys[lower + i] := keybuf[i];
        cards[lower + i] := cardbuf[i];
    end;
end;

procedure THand.Sort(keyfunc: TCardKeyFunc);
var
    cardbuf: TCardArray;
    keybuf, keys: TKeyArray;
    i: integer;
begin
    keys.setlength(cards.length);
    keybuf.setlength(cards.length);
    for i := 0 to cards.length - 1 do
        keys[i] := keyfunc(cards[i]);
    Sort(cardbuf, keybuf, keys, 0, cards.length);
end;

function _GetScore(card: TCard): integer;
begin
    result := card.GetScore;
end;

procedure THand.SortByRank;
begin
    Sort(@_GetScore);
end;

function _GetAltScore(card: TCard): integer;
begin
    result := card.GetAltScore;
end;

procedure THand.SortBySuit;
begin
    Sort(@_GetAltScore);
end;

end.
```

Listing 3: UHand.pas

```pascal
{$MODE OBJFPC}

```

```pascal
unit UUIQuerier;

interface

type
    IUIQuerier = interface
        function GetInt(msg: string): integer;
        function GetString(msg: string): string;
        procedure log(msg: string);
    end;

end.
```

Listing 4: UUIQuerier.pas

```pascal
{$MODE OBJFPC}

unit UGame;

uses UPlayer, UCard, UUIQuerier;

interface

type
    IGame = interface
        function GetGlobalState: string;
        function GetPrivateState(i: integer): string;
        function GetHelp: string;
        function GetCurrentPlayer: integer;
        procedure HandleTurn(querier: IUIQuerier);
        constructor UICreate(querier: IUIQuerier);
    end;

    TPestenGame = class(IGame)
    protected
        players: array of TPestenPlayer;
        num_packs: integer;
        card_pack: TPack;
        top_discard: TCard;
        suit_exemption: integer;
        curr_player_no, original_game_start: integer;
        history: array of string;
        history_start: integer;
        two_in_play: boolean;
        cur_two_rank, cur_two_suit, cur_two_acc: integer;
        curr_direction: integer;
        constructor Create(n_players, start_player, n_packs: integer);
        procedure WriteHistory(s: string);
        procedure HandleNormal(card: TCard; querier: IUIQuerier);
        procedure HandleTwo(card: TCard);
        procedure AdvanceSteps(steps: integer);
        procedure FreeAll;
    public
        function GetGlobalState: string;
        function GetPrivateState(i: integer): string;
        constructor UICreate(querier: IUIQuerier);
        procedure HandleTurn(querier: IUIQuerier);
        procedure HandlePickup;
        procedure HandleCardPlay
        function CardValid(card: TCard): boolean;
    end;

implementation

constructor TPestenGame.UICreate(querier: IUIQuerier);
```

```
var
    i: integer;
begin
    Create(querier.GetInt('How many players?'),
           querier.GetInt('Which player number deals?'),
           querier.GetInt('How many packs?'));
end;

constructor TPestenGame.Create(n_players, start_player, n_packs: integer);
var
    i: integer;
begin
    two_in_play := false;
    players.SetLength(n_players);
    history.SetLength(n_players * 2);
    history_start := 0;
    direction := 1;
    for i := 0 to history.length do
        WriteHistory('Game start');
    num_packs := n_packs
    card_pack := TPack.Create(n_packs);

    HandleCardPlay(card_pack.Deal);

    for i := 0 to players.length - 1 do
        players[i] := TPestenPlayer.Create(card_pack);

    curr_player_no := start_player;
    original_game_start := start_player;
end;

function GetGlobalState: string;
var
    i: integer;
begin
    result := 'history:' + #10;
    for i := history_start to history_start + history.length() - 1
        result := result + history[i mod history.length()] + #10;
    result := result + 'Top of discard is '
                     + discard_pile[num_discarded].GetString
                     + #10;
end;

function TPestenGame.GetPrivateState(i: integer): string;
begin
    result := players[i].GetState;
end;

function TPestenGame.GetCurrentPlayer: integer;
begin
    result := curr_player_no;
end;

procedure TPestenGame.HandleTurn(querier: IUIQuerier);
var
    user_card: string;
begin
    user_card := querier.GetString('What card would you like to play?');

    if user_card = 'take' then
        HandlePickup
    else begin
        if not players[curr_player_no].has_cardstring(user_card) then begin
            querier.log('This card is not in your hand');
            HandleTurn(querier);
        end else if not CardValid(player[curr_player_no].PeekCard(user_card)) then
            querier.log('This card is not valid to play')
            HandleTurn(querier);
        else
            HandleCardPlay(players[curr_player_no].PlayCard(user_card));
    end;
end;

function TPestenGame.CardValid(card: TCard): boolean;
begin
    if top_discard.GetRank = 10 then
        result := (card.Rank = 10) or (card.Suit = suit_exemption)
    else
        result := (card.Rank = top_discard.GetRank)
               or (card.Suit = top_discard.GetSuit);
end;

procedure TPestenGame.WriteHistory(s: string);
begin
    history[history_start] := s;
    history_start := (history_start + 1) mod history.length;
end;

procedure TPestenGame.HandlePickup;
begin
    if two_in_play then begin
        WriteHistory(Format('Player %d picks up %d cards', [curr_player_no, cur_two_acc]));
        players[curr_player_no].pickup(card_pack, cur_two_acc);
        two_in_play := false;
    end;
        WriteHistory(Format('Player %d picks up a card', [curr_player_no]));
        players[curr_player_no].pickup(card_pack);
end;

procedure TPestenGame.HandleCardPlay(card: TCard, querier: IUIQuerier);
begin
    if Player.GetCards = 1 then begin
        WriteHistory(Format('Player %d wins', [curr_player_no]));
        FreeAll;
        if querier.GetBool('Do you want to continue playing?') then
            Create(players.length, original_game_start, card_pack.length div 52)
        else
            raise EGameEnded.Create('Game is over');
    end;

    WriteHistory(Format('Player %d playrs a %s', [curr_player_no, card.GetString]));
    if two_in_play then
        HandleTwo(card)
    else
        HandleNormal(card, querier);
end;

procedure AdvanceSteps(steps: integer);
begin
    curr_player_no := proper_mod(curr_player_no + curr_direction * steps, players.length);
end;

procedure TPestenGame.HandleTwo(card: TCard);
begin
    if card.GetRank = cur_two_rank then
        cur_two_acc := cur_two_acc + cur_two_rank + 1
    else if (card.GetRank = cur_two_rank + 1)
        and (card.GetSuit = cur_two_suit) then begin
        inc(cur_two_rank);
        cur_two_acc := cur_two_acc + cur_two_rank + 1;
```

```pascal
    end;
    AdvanceSteps(1);
end;


procedure TPestenGame.HandleNormal(card: TCard, querier: IUIQuerier);
begin
    case card.GetRank in
        1: begin
            two_in_play := true;
            cur_two_rank := 1
            cur_two_suit := card.GetSuit;
            cur_two_acc := 2;
            AdvanceSteps(1);
        end;
        6: begin
            WriteHistory(Format('Player %d gets another turn', [curr_player_no]));
        end;
        7: begin
            WriteHistory(Format('Player %d skips a turn', [(curr_player_no + 1) mod players.
    ↪ length]));
            AdvanceSteps(2);
        9: begin
            WriteHistory(Format('Play goes back one turn', [(curr_player_no + 1) mod players
    ↪ .length]));
            AdvanceSteps(-1);
        10: begin do
                suit_exemption := querier.GetInt(
                        'What suit do you want to make it (ref:♠♣♡♢)');
            while not suit_exemption in [0..3];
            WriteHistory('Player %d sets suit to %s', [curr_player_no, suits[suit_exemption
    ↪ ]]);
        12: begin
            curr_direction := -curr_direction;
            AdvanceSteps(1);
        end;
    end;
end;

end.

{Things that aren't implemented:
    - jokers
    - runs
    - autoselection/ list of choices for user
    - better ui using ncurses, or a gui
        - which would ideally require proper XML communication
    - declaration of last card mechanism
}
```

Listing 5: UGame.pas

```pascal
{$MODE OBJFPC}

unit UUI;

uses UGame, UUIQuerier;

interface

type
    {User interface interface}
    IUI = interface(IUIQuerier)
        procedure DisplayText(txt: string);
        procedure ClearScreen;
        function AskPassword(msg: string): string;
    end;

    {Plain ansi terminal implementation of a UI}
    TTextUI = class(IUI)
    private
    public
        function GetInt(msg: string): integer;
        function GetString(msg: string): string;
        procedure log(msg: string);
        procedure DisplayText(txt: string);
        procedure ClearScreen;
        function AskPassword(msg: string): string;
    end;

implementation

function TTextUI.GetInt(txt: string): integer;
begin
    writeln(txt);
    write('Enter integer > '); readln(result);
end;

function TTextUI.GetString(txt: string): integer;
begin
    writeln(txt);
    write('Enter text > '), readln(result);
end;

procedure TTextUI.log(msg: string);
begin
    writeln('(game engine) ' + msg);
end;

procedure TTextUI.DisplayText(txt: string);
begin
    writeln(txt);
end;

procedure TTextUI.ClearScreen(txt: string);
begin
    {Ansi escape code to clear terminal}
    write(#27 + '[1;1H');
end;

procedure AskPassword(msg: string): string;
begin
    writeln(msg);
    write('Enter password > '); readln(result);
end;

end.
```

Listing 6: UUI.pas

```
unit UGameHandler;

uses UUI;

interface

type
    TGameHandler = class
    private
        UI: IUI;
        game_engine: IGame;
    public
        constructor Create(ui_var: IUI; game_var: IGame);
        procedure Play;
    end;

implementation
```
Listing 7: UGameHandler.pas